UnHash

2021

NMX Token and Staking Smart Contracts

security audit

Table of Contents

Part 1 Introduction

General Provisions

NMX Token and Staking Smart Contracts describe mechanics of the NMX token, it’s distribution and Uniswap liquidity pool staking.

UnHash was approached by Nominex to provide a security audit of the token and staking smart contracts.

Scope of the audit

The scope of the audit are the following smart contracts:

https://github.com/nominex/eth-smart-contracts/tree/3e70d2a05501d53a30d0bd149daeac7d12c8a555/contracts, git commit: 3e70d2a05501d53a30d0bd149daeac7d12c8a555.

Classification of Issues

CRITICAL:

Bugs leading to Ether or token theft, fund access locking or any other loss of Ether/tokens to be transferred to any party (for example, dividends).

MAJOR:

Bugs that can trigger a contract failure. Further recovery is possible only by manual modification of the contract state or replacement.

WARNINGS:

Bugs that can break the intended contract logic or expose it to DoS attacks.

COMMENTS:

Other issues and recommendations reported to/ acknowledged by the team.

Security Assessment Methodology

Security assessment is a multi-stage process. Each stage is further divided into multiple sub-stages and check routines. Eventually, the code under inspection is checked numerous times.

The unique UnHash methodology is developed through the course of many years of security analysis, and it can be briefly described as follows.

Unbiased code inspection

Reverse-engineering of models behind the software. That is where most logical inconsistencies pop up.

Guided code inspection

Documentation and code-guided inspection. They make sure the code works as intended, verifying with the models from the previous step. Checking invariants.

Automated analysis

Use of the best tools in the field, as well as a proprietary analyzer.

Checklist analysis

Use of a checklist, built through the course of several years of in-field experience.

Project- and sphere-specific analysis

Part 3 Executive Summary

Nominex turned to UnHash with a request to audit the security of token smart contracts and staking. The audit was led by Alexey Makeev.

The audited code is an ERC20 token, as well as the mechanics of its minting and staking rewards. Minting new tokens occurs exclusively by the token contract itself in accordance with the parameters of several pools. The Direct pool has the maximum minting speed and maximum release size, that are set in the code. A number of other pools have varying minting rates and release volumes, that are set in code.

Staking means providing liquidity in the Uniswap pool NMX-USDT and blocking the received LP-tokens on the balance of the so-called Staking Service. For this kind of provision of liquidity, rewards are issued in the NMX, and there is also a referral system.

Key phases of the audit were conducted from 5 to 13 February 2021. Despite the fact that there are no fundamentally complex mechanics and calculations in the code, the audit required a lot of resources. Because almost the whole code works with tokens and there are many points and potential vulnerabilities that need to be addressed in such cases.

Among other things, the DeFi-specific attack vectors, effort with mathematics and various types of data were checked, and full-cycle simulation was carried out with the introduction of additional checks. The Project's high-quality test cases helped a lot with the last mentioned operation above.

The codebase itself is implemented on a high level. The main problems have been identified in blockchain-specific areas and they can be summarized the following way:

- problems of manipulation of the referral system
- oracle manipulation problems
- blocking of a part of issued tokens under certain conditions

In addition, several issues were identified in the fixes, and they were fixed. The use of "small" numeric types combined with the lack of overflow control can also be classified as dangerous practices. However, no actual overflows were identified.

The referral system and the oracle, as a part of it, were eventually removed from the code.

Issues summary

Issue type	Number
Critical	3
Major	2
Warning	9
Comment	10

It's clear that the project is working hard to save gas. With all other things being equal, this is good. But, on the other hand, this leads to more complex code and the appearance of errors.

We would like to give you a broad guideline. It is a good practice to write correctly functioning software covered with tests in the first place and then optimize in situ, starting with profiling, if it's necessary.

The key security factors of the system are the minimum number of state variables, incremental updating of staking balances, the absence of external calls to other contracts (we consider here the Uniswap contract of the pair is audited and verified), and an extensive testing base.

According to our analysis, the commit a491daf68a6cac15e76173f83efd0036ecefc328 does not have any vulnerabilities.

CRITICAL

01 StakingService.sol#L350-L352

Any NMX supplies accrued while `state.totalStaked` is zero will be stuck on the balance of the service forever, as there is no way to recover them. They will not be distributable via rewards as well.
We suggest adding some distribution mechanics for such supplies.
Fixed at 59587cb

02 LiquidityWealthEstimator.sol#L40

This price estimation algorithm can be easily and risk-free manipulated with a flash loan.

Consider the following example:

let `lpAmount / lpTotalSupply = 0.1,`

let the NMX price = 100 USDT,

let the initial pool content is (1000 USDT, 10 NMX).

The pool value is `1000 + 10 * 100 == 2000,` the function returns `0.1 * 2000 == 200.`

An attacker can borrow a flash loan of 9000 USDT and sell them to the pool, receiving 9 NMX (the fee omitted for simplicity's sake). The constant product still holds: `1000 * 10 == 10000 * 1.`

Now the function will return `0.1 * 10000 * 2 == 2000,` i.e., 10 times more.

After the reward calculation, the attacker buys back 9000 USDT and repays the loan, while having his wealth overestimated 10 times.

Note that somehow preventing flash loans would not solve the problem as sandwich attacks would still be possible, although not risk-free. More information on on-chain price estimation can be found here: https://uniswap.org/docs/v2/core-concepts/oracles/.

We suggest employing a TWAP-based oracle for wealth estimation. Fortunately, there is only one pair and `estimateWealth` calls are frequent.

Fixed at 9b6c5c1

03 StakingRouter.sol#L35

In case of `cumulativeShare < 1` some share of supplies, namely, `supply * (1 - cumulativeShare)`, will be stuck on the balance of the router forever as there is no way to recover it.

We recommend ensuring that `cumulativeShare == 1`.

Fixed at 6b889c7

MAJOR

01 StakingService.sol#L249-L261

Referral systems are prone to Sybil attacks. In this case, a staker can employ another account as a referrer (this account may also appoint the first account as their referrer). Moreover, just for the short duration of the `updateStateAndStaker` call the stacker can move a sufficient amount of funds (10 000 USDT at the moment) to the referrer account to get the maximum bonus added to his reward, essentially performing a sandwich attack.

A flash loan can be used to decrease the cost of such an attack to a minimum.

One may say that since the bonuses are paid from the direct pool out of thin air, the impact is low. However, the economic consequences of this attack can not be underestimated, as all holders of NMX get diluted.

The safest solution is to disable the referral mechanic altogether. We suggest redesigning the referral approach.

Fixed at 0b6609a

02 StakingService.sol#L262-L268

Similarly to the problem described above, anyone can set the referrer to any address to get a modest `referralBonus`. We suggest redesigning the referral approach.

Fixed at 0b6609a

WARNING

01 MintSchedule.sol#L180

In rare cases in the future the loop may reach the block gas limit, rendering the pool stalled. We recommend adding a version of the function which limits the number of loop iterations.

Fixed at f44253c

02 StakingService.sol#L116

If the `owner` issued an open approval to the staking service (when the amount is `uint256(-1)` or another high number), a certain form of a griefing attack is possible. Anyone can force the tokens of such `owner` to be staked. Make sure that this risk is acceptable.

Acknowledged

Client: Staking tokens does not imply negative consequences for such an owner. According to the contract, the owner can withdraw the staked tokens at any time, as well as the income received in the form of accrued Nmx tokens.

03 StakingService.sol#L281

The referrer can not claim his rewards until their referral is updated. Make sure this behaviour is acceptable.

Fixed at 0b6609a

04 DirectBonusAware.sol#L83

Some third-party contract which happens to be in a call chain can set the referrer of the tx.origin to any value. Usually, users do not call untrusted contracts. However, a call chain can be deep and contract interactions quite cumbersome. Plus, calling a contract with zero ether value may seem safe.

Make sure this behaviour is acceptable.

Fixed at 0b6609a

05 StakingRouter.sol#L42

Any supplies accrued and not distributed up to this moment by the current services are distributed among new services. We suggest issuing a `updatePendingSupplies` call before changing services configuration.

Fixed at f296ff0

06 LiquidityWealthEstimator.sol#L36

Because of the direct bonus existence, the actual max supply is about 233 million.

Fixed at 5717010

07 StakingRouter.sol#L45

We suggest checking for duplicates in the `addresses` parameter. Otherwise, the sum of the shares may effectively go over 1.

We also suggest checking that the addresses are non-zero.

Fixed at 4690efe

08 MintSchedule.sol#L9

Nmx.sol#L10

StakingRouter.sol#L9

StakingService.sol#L14

Most of the contracts are `Ownable.` That gives the `owner` address significant privileges. Ultimately, it can drain 20 000 tokens a day, or block the supply altogether. We advise using a multi-signature contract as the `owner,` as well as implementing a time lock for the most crucial parameter changes.

Acknowledged

09 StakingService.sol#L354

StakingService.sol#L243-L244

Given the current NMX supply parameters, there is no `uint128` overflow here. However, the code might be a couple of bits away from overflow. Since the `StakingService` contract is likely to evolve, that uneasy equilibrium might eventually be broken. We suggest performing these computations in the `uint256` domain.

Fixed at 27a4e6e

COMMENT

01 Nmx.sol#L10

Since the token is `Ownable`, we recommend adding `PausableByOwner` functionality which blocks any transfers and minting. It would greatly simplify token migration if such a need arises.

Acknowledged

02 MintSchedule.sol#L19

It makes no sense to use `uint64` instead of `int128` here, as no storage space will be saved. We suggest `using int128`. As a result, many type casts in the code would be omitted.

Fixed at ea8855f

03 MintSchedule.sol#L45

Here and below we suggest storing `poolShares` as `int128` since no storage space will be saved by using `uint64.` The current code risks truncating values. Extra checks may be added ensuring that each share is no greater than 1.

Fixed at 2e3cea7

04 Nmx.sol#L110

Here and below since the contract inherits from `Context` it makes sense to use `_msgSender()` instead of `msg.sender.`

Fixed at 9557c3e

05 Nmx.sol#L165-L170

The loop can be replaced by a `poolByOwner[newOwner]` lookup.

Fixed at c23b6d6

06 PausableByOwner.sol#L19

PausableByOwner.sol#L30

There is no need in the `whenNotPaused` and `whenPaused` checks, since they exist in Pausable.

Fixed at 45d99a5

07

StakingService.sol#L92

StakingService.sol#L102

StakingService.sol#L115

The `whenNotPaused` modifier can be moved to the private `_stakeFrom` implementation to avoid repetition.

Fixed at 6ae189a

08

StakingService.sol#L179-L180

StakingService.sol#L191

StakingService.sol#L198

StakingService.sol#L205

StakingService.sol#L243

StakingService.sol#L298

StakingService.sol#L346

StakingService.sol#L275

StakingService.sol#L131-L132

StakingService.sol#L281

StakingService.sol#L284

StakingService.sol#L289

StakingService.sol#L301

StakingService.sol#L353-L354

It’s good to use the `SafeMath` library to check for overflows and underflows. Although, that is no longer necessary in Solidity 0.8.0+.

Acknowledged

09

StakingService.sol#L161

StakingService.sol#L224

The data type of the `value` parameter differs from the one defined in the function type hashes. We recommend fixing the type hashes.

Fixed at 4ed5ab3

10

StakingService.sol#L298

StakingRouter.sol#L51

LiquidityWealthEstimator.sol#L30

We suggest exiting the execution as soon as it is known that the token value is zero. That prevents expensive external calls and meaningless zero-value event emissions.

Acknowledged

Part 5 **About UnHash**

About UnHash

UnHash is a security auditing and consulting company valuing software security as the top priority. It provides security assessment and consulting for the most innovative DLT software projects.

Contact us:

https://unhash.io

hello@unhash.io

Part 6 Disclaimer

UnHash does not guarantee any commercial, investment or other benefits from the use or launch of products based on the audited code. UnHash may be involved in the promotion of the code and/or products based on it only with its written consent.